第11章 引用和拷贝构造函数

本章内容

- 引用及其传值与传引用
- 拷贝构造函数
- 指向成员的指针

C++中的指针

- 指针与类型相关的问题
  - C++中的指针是从C语言中沿用而来，和C相比，C++中的指针对类型的要求更高。下面的代码在C语言中是正确的，而在C++中将产生编译错误：

```
Bird* b;
Rock* r;
void* v;
v = r;
b = v;
```

  - C++中必须通过显示地使用强制类型转换。

C++中的指针

- 指针与内存相关的问题
  - 悬垂指针（Dangling pointer）

```
int* arrayPtr1;
int* arrayPtr2 = new int[100];
arrayPtr1 = arrayPtr2;
delete [] arrayPtr2;
```

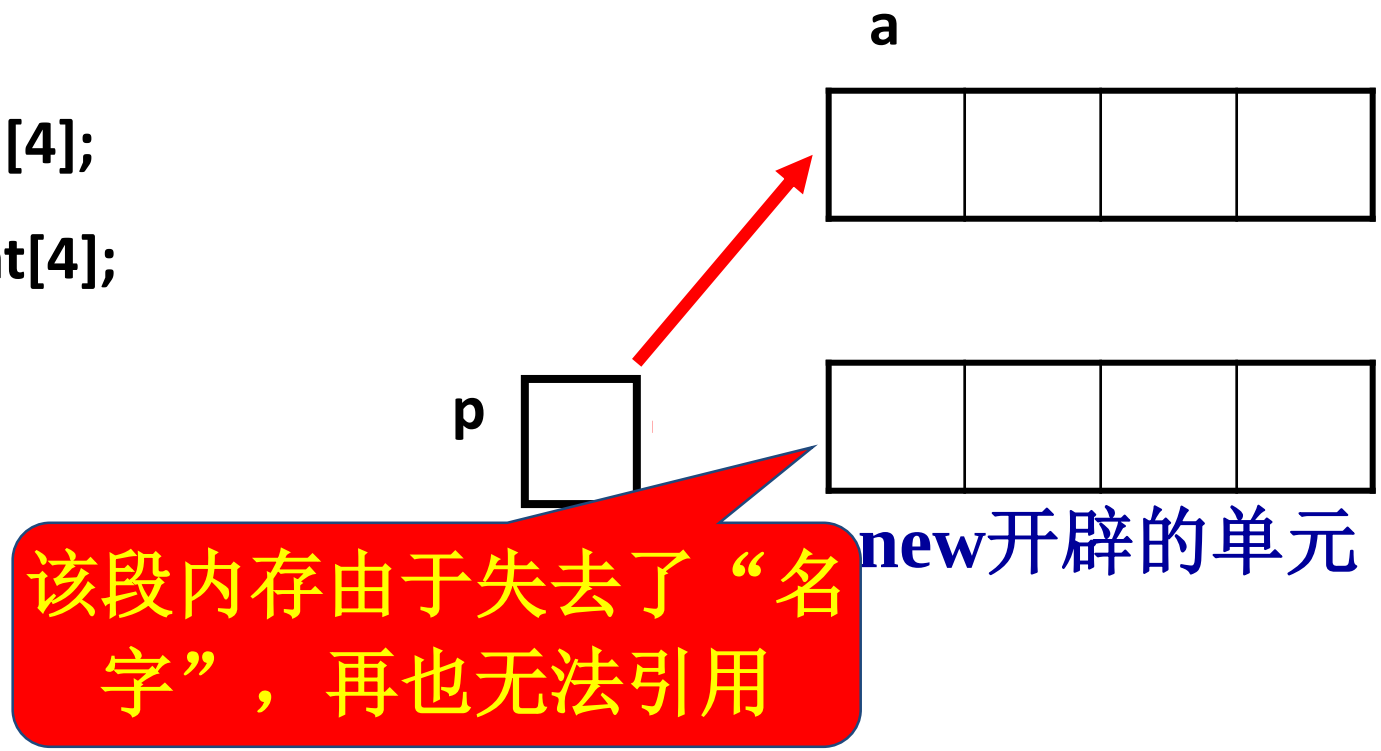
C++中的指针

- 内存泄漏：
  - 用new开辟的内存单元没有名字，指向其首地址的指针是引用其的唯一途径，若指针变量重新赋值，则用new开辟的内存单元就在内存中“丢失”了，别的程序也不能占用这段单元，直到重新开机为止。

```
int  * p, a[4];

p=new int[4];

p=a;
```



C++中的引用

```
// Ordinary free-standing reference:
int y;
int& r = y;
// When a reference is created, it must
// be initialized to a live object.
// However, you can also say:
const int& q = 12;  // (1)
// References are tied to someone else's storage:
int x = 0;          // (2)
int& a = x;         // (3)
int main() {
    cout << "x = " << x << ", a = " << a << endl;
    a++;
    cout << "x = " << x << ", a = " << a << endl;
} ///:~
```

C++中的引用

- 对变量起另外一个名字（别名），这个名字称为该变量的引用。
  - <类型> &<引用变量名> = <原变量名>;
- 其中原变量名必须是一个已定义过的变量，如：
  - int max ;
  - int &refmax=max;
- refmax并没有重新在内存中开辟单元，只是引用max的单元。max与refmax在内存中占用同一地址，即同一地址两个名字。

C++中的引用

- 可以认为，引用是特殊的指针，指针指示内存中的地址，而引用指示内存中的对象。
- 对引用的操作就是对被引用的变量的操作
- 使用引用时的注意：
  - 引用被声明的同时，必须被初始化
  - int &refmax; 错误，没有具体的引用对象
  - int &refmax=max; 正确，max是已定义过的变量
    - 引用类型变量的初始化值不能是一个常数

C++中的引用

- 使用引用有如下的限制
  - 不能像指针一样进行运算
  - 一旦一个引用被初始化指向一个对象，就不能改变为对另一个对象的引用。
  - 不可能有NULL引用(因为引用必须被初始化)。

函数中的引用

- 引用常见于函数的参数和返回值中。尽管效果与指针相同，但是引用具有更加清晰的语法。
- 如果从函数中返回一个引用，该引用必须为一个有效的地址，即其生命周期不应该在函数返回后就结束。
- 把函数定义为引用类型，这时函数的返回值即为某一变量的引用（别名），因此，它相当于返回了一个变量，所以可对其返回值进行赋值操作。这一点类同于函数的返回值为指针类型。

函数中的引用

```
int* f(int* x) {
    (*x)++;
    return x; // Safe, x is outside this
scope
}
int& g(int& x) {
    x++; // Same effect as in f()
    return x; // Safe, outside this scope
}

int& h() {
    int q;
    //!  return q;  // Error
    static int x;
    return x; // Safe, x lives outside this
scope
}
```

```
int main() {
    int a = 0;
    f(&a); // Ugly (but explicit)
    g(a);  // Clean (but hidden)
} ///:~
```

常量引用

- 如果一个参数是引用类型，并且该参数有可能需要传递常量，则应该使用常量引用。

```
void f(int&) {}
void g(const int&) {}

int main() {
    //!  f(1); // Error
    g(1);
} ///:~
```

指针引用:传统方法

- 在C语言中，如果想改变参数中指针本身而不是它所指向的内容，该函数可声明为以下的形式：
  - void f(int**)
- 但传递时，必须获得指针的地址：
  - int i = 47;
  - int* ip = &i;
  - f(&ip);

指针引用：C++中的引用

```
//C11: ReferenceToPointer.cpp

void increment(int*& i) { i++; }

int main() {
    int* i = 0;
    cout << "i = " << i << endl;
    increment(i);
    cout << "i = " << i << endl;
} ///:~
```

指针与引用的区别

- 1、指针是通过地址间接访问某个变量，而引用是通过别名直接访问某个变量。

- 2、引用必须初始化，而一旦被初始化后不得再作为其它变量的别名。

- 当&a的前面有类型符时（如int &a），它必然是对引用的声明；如果前面无类型符（如cout<<&a）,则是取变量的地址。

```
class Test
{
    public static void main(String[] args)
    {
        StringBuffer s= new StringBuffer("good");
        StringBuffer s2=s;
        s2.append(" afternoon.");
        System.out.println(s);
    }
}
```

输出：

good afternoon

参数传递的准则

- 传值与传引用
  - 在Java中，程序员几乎不需要知道有传值和传引用的区分，因为程序员对此没有选择的余地。
  - 在C++中，对于函数的每一个参数，都需要确定究竟是以值的方式还是以引用的方式传递。
- 给函数传递参数时，通常是通过常量引用来传递，这种简单的习惯可以大大提高执行效率。
- 只有极少数的情况需要需要以传值方式传递对象。

引用作为形参，实参是变量而不是地址，这与指针变量作形参不一样。

形参为整型引用

```
void  change(int &x,  int &y)
{   int   t;
    t=x;  x=y;   y=z;
}
void main(void)
{   int  a=3,b=5;
   change(a,b);  //实参为变量
  cout<<a<<'\t'<<b<<endl;
}
```

形参为指针变量

```
void  change(int *x,  int *y)
{   int   t;
    t=*x;  *x=*y;   *y=z;
}
void main(void)
{   int  a=3,b=5;
   change(&a,&b); //实参为地址
  cout<<a<<'\t'<<b<<endl;
}
```

引用小结

- 可以认为，引用是特殊的指针，指针指示内存中的地址，而引用指示内存中的对象。
- 引用常用于函数参数的传递。
- 使用引用有如下的限制
  - 引用被声明的同时，必须被初始化
  - 一旦一个引用被初始化指向一个对象，就不能改变为对另一个对象的引用。

拷贝构造函数

- 对于基本数据类型，如字符，整数，浮点数，传值和返回值的机制是什么？
- 对于大对象而言，传值和返回值的实现机制是什么？

基本数据类型的传值和返回值

```
int f(int x, char c);
int g = f(a,b);
```

```
push b
push a
call f()
add sp,4
mov g, register a
```

大对象的传值和返回值

下面的代码中，大对象会如何传递和返回？

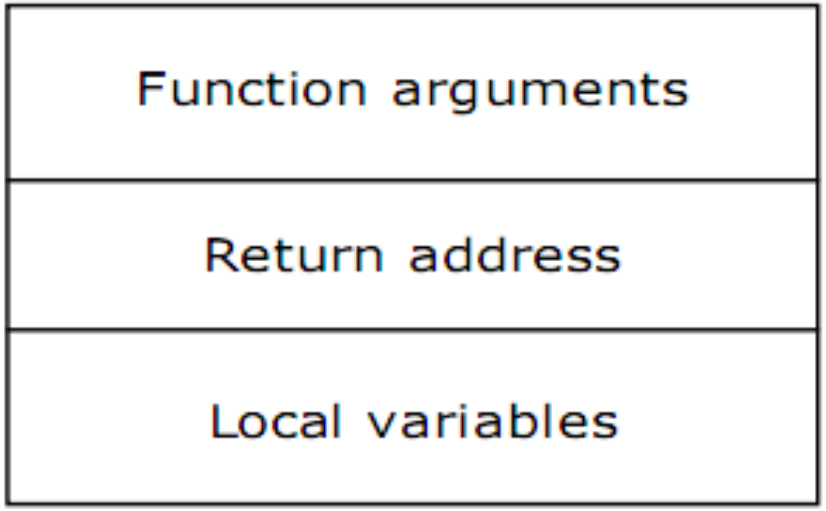
```
//: C11:PassingBigStructures.cpp
struct Big {
  char buf[100];
  int i;
  long d;
} B, B2;

Big bigfun(Big b) {
  b.i = 100; // Do something to the argument
  return b;
}

int main() {
  B2 = bigfun(B);
} ///:~
```

函数调用框架

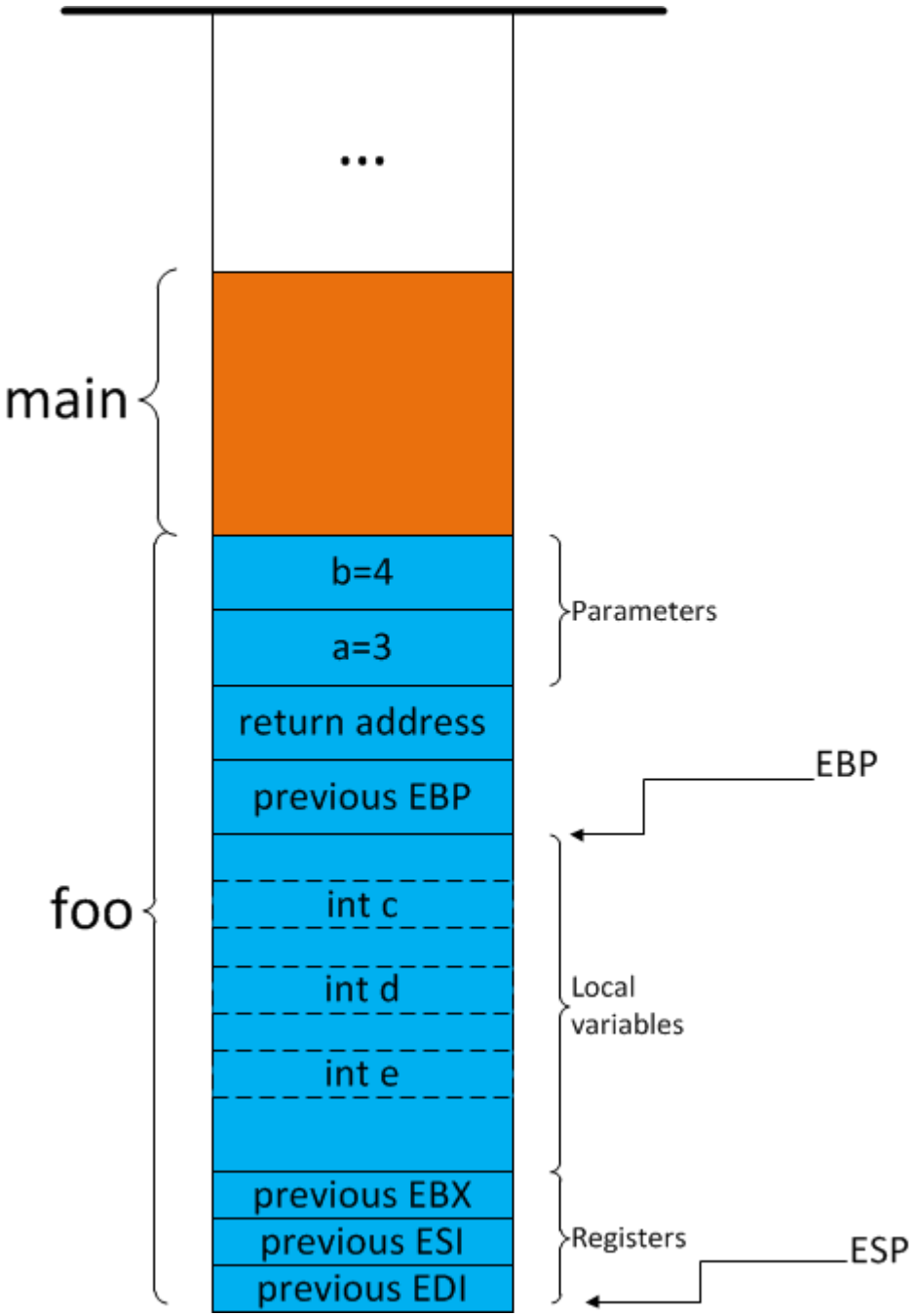
- 在call后的栈框架（函数已经为局部变量分配了存储单元）

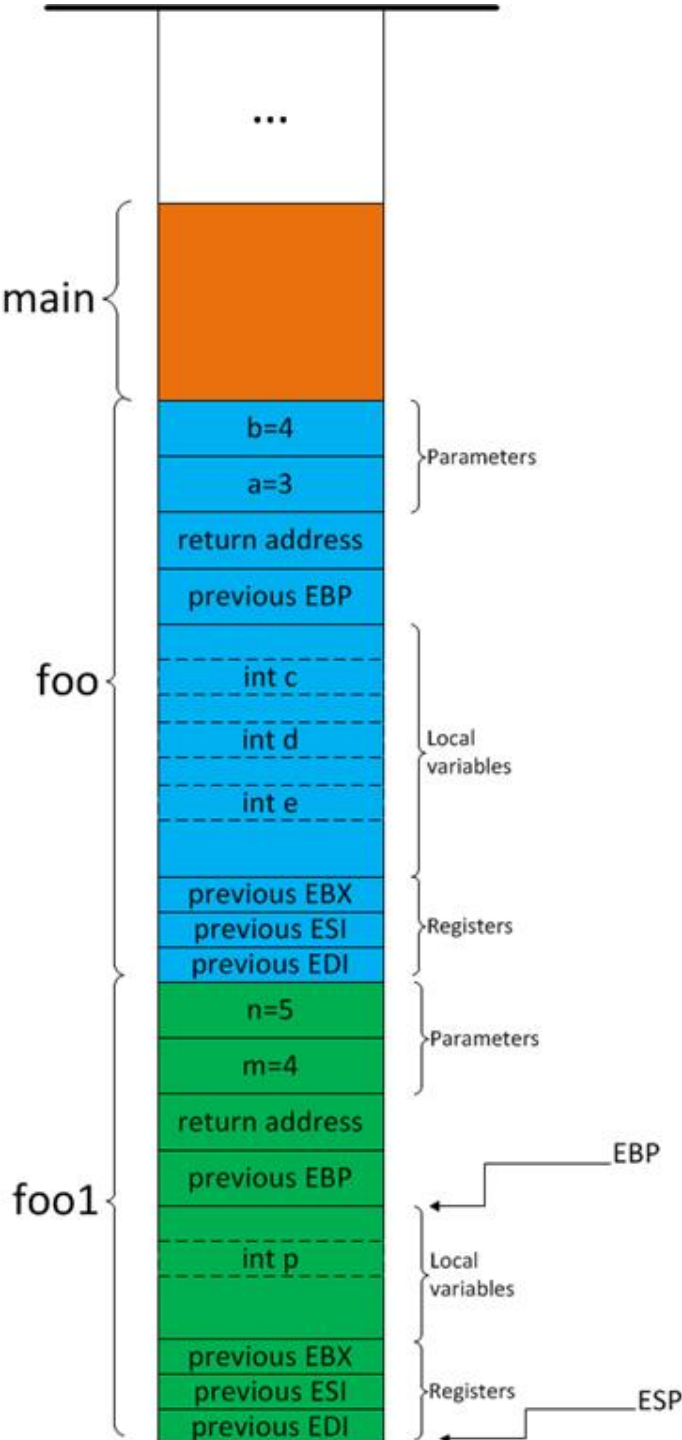


函数调用框架

```
int foo1(int m, int n)
{
   int p=m*n;
   return p;
}
int foo(int a, int b)
{
   int c=a+1;
   int d=b+1;
   int e=foo1(c,d);
   return e;
}

int main()
{
   int result=foo(3,4);
   return 0;
}
```


...
main
b=4
a=3
return address
previous EBP
foo
int c
int d
int e
previous EBX
previous ESI
previous EDI
Parameters
EBP
Local
variables
Registers
ESP


...
main
b=4
a=3
return address
previous EBP
foo
int c
int d
int e
previous EBX
previous ESI
previous EDI
Parameters
Local
variables
Registers
n=5
m=4
return address
previous EBP
foo1
int p
previous EBX
previous ESI
previous EDI
Parameters
EBP
Local
variables
Registers
ESP

函数调用框架

- 如果将返回值放在栈中，由调用方来取，有什么问题？
- 将返回值放在全局堆中呢？
- 中断，导致重入(re-entrancy)

函数调用框架

- ISR中断服务程序负责存储和还原所使用的所有寄存器，保护和恢复现场。

- 由于不能触及返回地址以上的任何部分，所以函数必须在返回地址以下将返回值压栈。return执行时，堆栈指针必须指向返回地址。在return之前，函数必须将堆栈指针向上移动，这便清除了所有局部变量。此时万一发生中断怎么办？ISR向下移动堆栈指针，保存返回地址和局部变量，会覆盖掉返回值。

函数调用框架

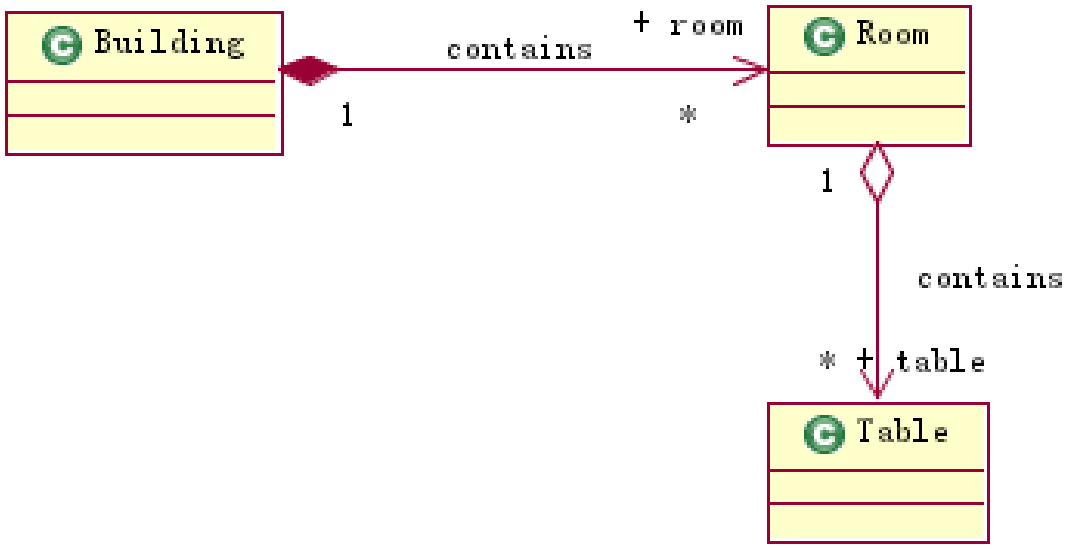
- 将返回值放在寄存器中
  - 如果返回值等于4字节，函数将把返回值赋予EAX寄存器，通过EAX寄存器返回。例如返回值是字节、字、双字、布尔型、指针等类型，都通过EAX寄存器返回。
  - 如果返回值等于8字节，函数将把返回值赋予EAX和EDX寄存器，通过EAX和EDX寄存器返回，EDX存储高位4字节，EAX存储低位4字节。例如返回值类型为__int64或者8字节的结构体通过EAX和EDX返回。
  - 如果返回值为double或float型，函数将把返回值赋予浮点寄存器，通过浮点寄存器返回。
- 如果返回值是一个大于8字节的数据，将如何传递返回值呢？

返回值的解决方案

- 将返回值的地址作为参数进栈，函数返回时，将作为局部变量的返回值的所有内容复制到该地址所指向的区域；
- 默认情况下，C++使用位拷贝的机制完成这个复制的过程，实现对该对象的初始化；
- 然而，是否有问题？
  - C11:HowMany.cpp

位拷贝的问题

- A是B的物理或逻辑组成部分
- A被物理或逻辑地包含在B中



C++的拷贝构造函数

- 可以在定义一个对象的时候用另一个对象为其初始化，即构造函数的参数是另一个对象的引用，这种构造函数常为完成拷贝功能的构造函数。
- 完成拷贝功能的构造函数的一般格式为：

```
ClassName::ClassName(ClassName  &<变量名>)
{   ……
//   函数体完成对应数据成员的赋值
}
```

```
class  A{
        float   x,y;
public:
  A(float a=0, float b=0){x=a;  y=b;cout<<"调用了构造函数\n";}
  A(A &a)
      { x=a.x;y=a.y;cout<<“调用了完成拷贝功能的构造函数\n”; }
  void  Print(void){  cout<<x<<'\t'<<y<<endl;  }
  ~A() {  cout<<"调用了析构函数\n";   }
};
void main(void)
{   A   a1(1.0,2.0);
    A   a2(a1);
    a1.Print();
    a2.Print();
}
```

用已有的对象中的数据为新创建的对象赋值

```
调用了构造函数
调用了完成拷贝功能的构造函数
1               2
1               2
调用了析构函数
调用了析构函数
```

C++的拷贝构造函数

- 如果没有定义完成拷贝功能的构造函数，编译器自动生成一个隐含的完成拷贝功能的构造函数，依次完成类中对应数据成员的拷贝

- 问题是什么？

```
class  A{
        float   x,y;
public:
  A(float a=0, float b=0){x=a;  y=b;cout<<"调用了构造函数\n";}
  void  Print(void){  cout<<x<<'\t'<<y<<endl;  }
  ~A() {  cout<<"调用了析构函数\n";   }
};
void main(void)
{   A   a1(1.0,2.0);
    A   a2(a1);
    A   a3=a1;//可以这样赋值
    a1.Print();
    a2.Print();
    a3.Print();
}
```

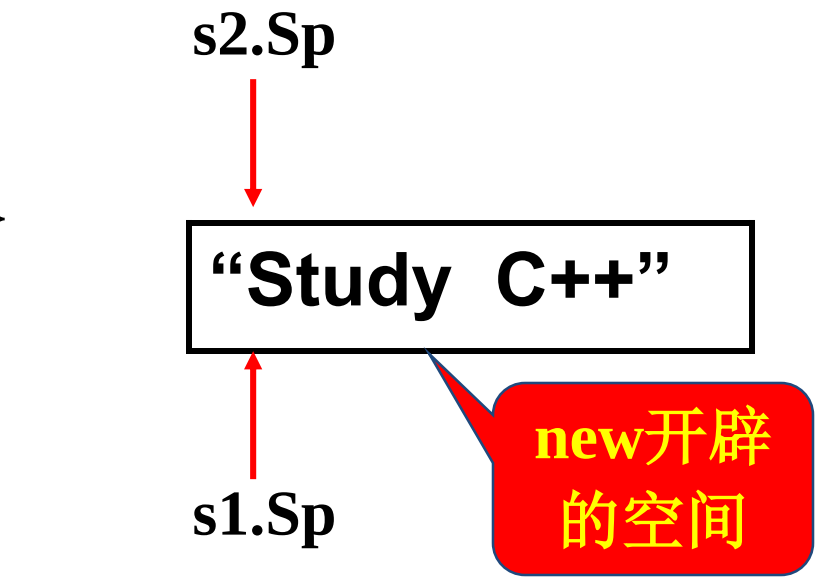
隐含了拷贝的构造函数

```
调用了构造函数
1       2
1       2
1       2
调用了析构函数
调用了析构函数
调用了析构函数
```

```
class  Str{
   int Length;   char  *Sp;
public:
   Str(char *string){
   if(string){Length=strlen(string);
      Sp=new char[Length+1];
      strcpy(Sp,string);          }
      else          Sp=0;
  }
void Show(void){cout<<Sp<<endl;}
~Str(){if(Sp)  delete []Sp;    }
};
void main(void)
{          Str s1("Study C++");
           Str s2(s1);
           s1.Show ();     s2.Show ();
}
```

隐含的拷贝构造函数为：

```
Str::Str(Str &s)
{          Length=s.Length;
           Sp=s.Sp;
}
```

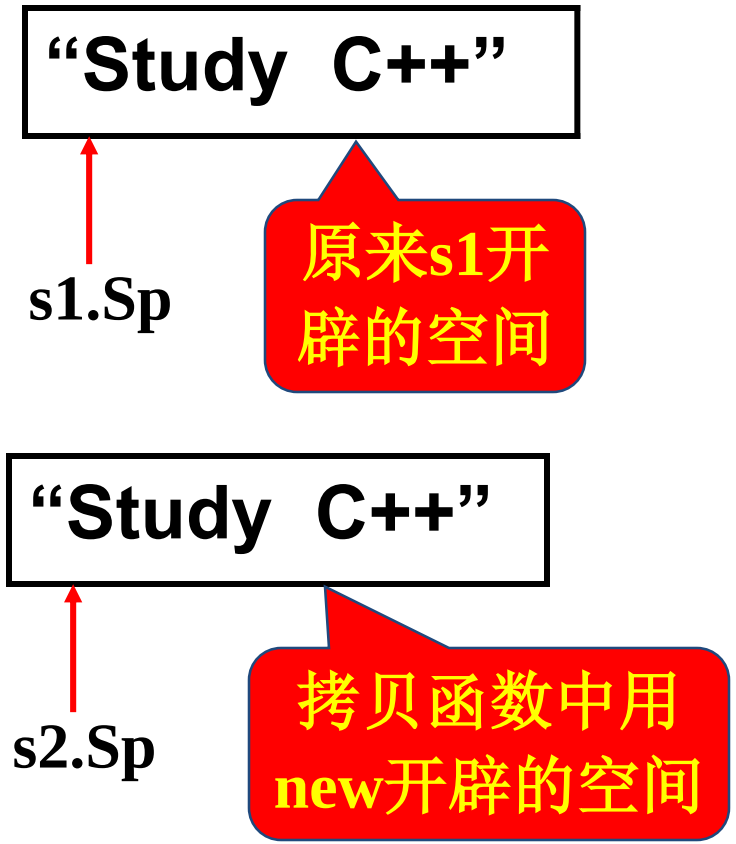


同一空间释放两次，造成运行错误。

在这种情况下，必须要定义完成拷贝功能的构造函数。

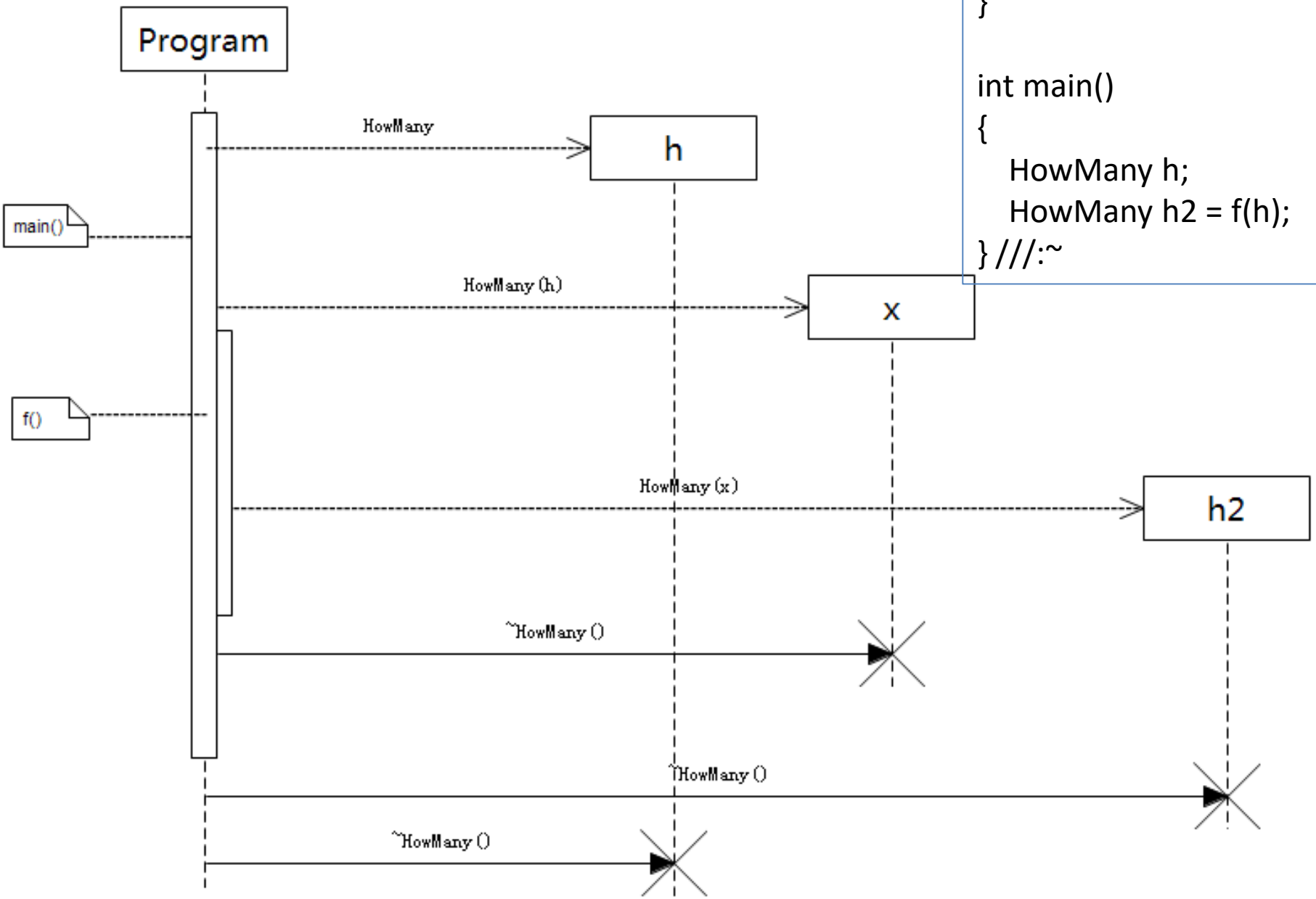
```
Str::Str(Str &s){
  if(s.Sp){
        Length=s.Length ;
       Sp=new char[Length+1];
        strcpy(Sp,s.Sp);
        }
        else Sp=0;
}
```

```
Str s2(s1);
```



```
HowMany f(HowMany x){
        return x;
}

int main()
{
   HowMany h;
   HowMany h2 = f(h);
} ///:~
```



临时对象

```
class Person{
}

Person func()
{
    Person p;
    return p;
}

int main() {
    Person xp= func();
  } ///:~
```

防止拷贝构造

- 如果不需要传递和返回值，不需要将一个类型声明为值对象，那么可以不实现拷贝构造函数。
- 事实上，传递值和返回值是一个非常容易出错的过程，一个好的习惯是，**避免传值和返回值的情况**。

防止拷贝构造

- 可以通过将拷贝构造函数声明为私有，而防止传值和返回值的操作。
- 除非成员函数和友元函数需要执行按值传递方式的传递，否则编译错误

  – C11: NoCopyConstruction.cpp

传递参数的约定

- 引用的语法比指针语法清晰，但却使得含义变得模糊。
- 如果要改变参数，建议使用传指针的方式。
- 所有传引用的参数都声明为const.

拷贝构造函数小结

- 如果一个类需要传值和返回值，编译器需要调用该类的拷贝构造函数。
- 可以自定义拷贝构造函数，如果没有自定义的拷贝构造函数，编译器会自动生成一个按位拷贝的拷贝构造函数（糟糕）。
- 出于效率和降低复杂性的考虑，尽可能采用传指针（需修改返回值）和传参数引用（无需修改返回值），同时在语法上防止拷贝构造。

指向成员的指针

- 指向数据成员的指针
- 指向函数成员的指针

指向数据成员的指针

```
struct Simple { int a; };
int main() {
  Simple so, *sp = &so;
  sp->a;
  so.a;
} ///:~
```

指向int的指针以如下的方式定义：

```
int * pInt;
```

如何定义一个指向Simple中的int a的指针?

```
int  Simple::*pInt = &Simple::a;
```

```
<type> PointName = &ClassName::member;
```

指向数据成员的指针

- 例：
  - C11: PointerToMemberData.cpp

```
class Data {
public:
  int a, b, c;
  void print() const {
    cout << "a = " << a << ", b = " << b << ", c = " << c << endl;
  }
};

int main() {
  Data d, *dp = &d;
  int Data::*pmInt = &Data::a;
  dp->*pmInt = 47;
  pmInt = &Data::b;
  d.*pmInt = 48;
  pmInt = &Data::c;
  dp->*pmInt = 49;
  dp->print();
} ///:~
```

指向成员函数的指针

- 指向函数的指针：
- <type> (ClassName:: *PointName)(<ArgsList>);

  int (*fp) (float);
- 指向成员函数的指针：

```
class Simple2 {
public:
  int f(float) const { return 1; }
};
int (Simple2::*fp)(float) const;
int (Simple2::*fp2)(float) const = &Simple2::f;
int main() {
  fp = &Simple2::f;
} ///:~
```

指向成员函数的指针

```
class Widget {
public:
  void f(int) const { cout << "Widget::f()\n"; }
  void g(int) const { cout << "Widget::g()\n"; }
  void h(int) const { cout << "Widget::h()\n"; }
  void i(int) const { cout << "Widget::i()\n"; }
};

int main() {
  Widget w;
  Widget* wp = &w;
  void (Widget::*pmem)(int) const =
&Widget::h;
  (w.*pmem)(1);
  (wp->*pmem)(2);
} ///:~
```

```
class Widget {
  void f(int) const { cout << "Widget::f()\n"; }
  void g(int) const { cout << "Widget::g()\n"; }
  void h(int) const { cout << "Widget::h()\n"; }
  void i(int) const { cout << "Widget::i()\n"; }
  enum { cnt = 4 };
  void (Widget::*fptr[cnt])(int) const;
public:
  Widget() {
    fptr[0] = &Widget::f; // Full spec required
    fptr[1] = &Widget::g;
    fptr[2] = &Widget::h;
    fptr[3] = &Widget::i;
  }
  void select(int i, int j) {
    if(i < 0 || i >= cnt) return;
    (this->*fptr[i])(j);
  }
  int count() { return cnt; }
};

int main() {
  Widget w;
  for(int i = 0; i < w.count(); i++)
    w.select(i, 47);
} ///:~
```

注意事项

- 指向类中成员的指针变量不是类中的成员，这种指针变量应在类外定义。
- 使用这种指针变量来调用成员函数时，必须指明调用哪一个对象的成员函数，这种指针变量是不能单独使用的。用对象名引用。
- 由于这种指针变量并不是类的成员，所以使用它只能访问对象的公有成员。若要访问对象的私有成员，必须通过类中的其它公有成员函数来实现。
- 当用这种指针指向静态的成员函数时，可直接使用类名而不要列举对象名。

指向成员的指针小结

- 指向成员的指针和普通指针一样：可以在运行的时候访问存储单元中的数据或函数，区别是成员指针只能与类成员一起工作。
- 通过使用指向成员的指针，我们的程序可以设计为能够在运行时灵活地改变行为。